Coding Rules Overview

Sean Barnum, Cigital, Inc. [vita[1]]

2005-10-03

Coding rules are representations of knowledge, gained from real-world experiences, about potential vulnerabilities that exist in programming languages like C and C++. As we create and use software with a given coding environment, we discover and learn about many vulnerabilities that exist in this environment, how to recognize whether they crop up in our code, and what to do to fix them.

As an individual interested in learning more about *Building Security In* to software, you may ask yourself “What are coding rules and why should I care about them?”

Coding rules are representations of knowledge, gained from real-world experiences, about potential vulnerabilities that exist in programming languages like C and C++. As we create and use software with a given coding environment, we discover and learn about many vulnerabilities that exist in this environment, how to recognize whether they crop up in our code, and what to do to fix them.

While developing this expertise certainly brings much higher levels of quality and security to code written by the experienced practitioner, the reality is that such expertise is in very short supply and this knowledge is nontrivial to convey. Recognized experts are typically leveraged to help a broader base of less experienced developers through the practice of security code reviews.

Put into play this way, these rules become a checklist that the practitioner looks for when performing a security review of a piece of code. An example is the use of the C function call *strcpy()* that, as it is typically used, opens a piece of software up to significant danger of exploit through a buffer overflow. A skilled security reviewer checks for uses of this function and replaces them with a more secure function (e.g., making use of the STL) or at least verifies that in each case proper buffer management and bounds checking occurs.

Even with this kind of expert/mentor approach, there is still far too much code being written for the practice to scale. Two actions are required to improve the efficiency and effectiveness of this process. First, the knowledge of these rules needs to be transformed from its implicit state in the minds of experts into a more explicit form that allows it to be shared. Second, because this kind of work is tedious, boring, and difficult to perform, automated tools are needed to review source code and apply these rules. Let's face it, experts hate manually digging through thousands of lines of code looking for these kinds of issues. Automated tools are able to review code and apply rules at a faster rate, at a higher level of quality, and to larger code bases than any human could in a purely manual fashion.

The good news is that both of these things are taking place.

Evolution of Coding Rules

History of Rule Coverage

Coding rules in explicit form have evolved rapidly in their coverage of potential vulnerabilities. Before Bishop and Dilger's 1996 work on race conditions in file access [Bishop 96][2], explicit coding rulesets, if

[1] daisy:35 (Barnum, Sean)

[2] #refs

they existed at all, only existed as checklist documents of ad hoc information authored, managed, and typically not widely shared by experienced software security practitioners. Bishop and Dilger's tool was one of the first recognized attempts to capture a ruleset (in this case, a limited set of rules covering potential race conditions in file accesses using C on UNIX systems) and automate its application through lexical scanning of code. For the next four years, a lot of thinking and research was done in the area, but no other tools and accompanying rulesets emerged to push things forward.

This changed in early 2000 with the release of ITS4[3], a tool whose ruleset also targeted C/C++ code but went beyond the single-dimensional approaches of the past to cover a broad range of potential vulnerabilities in 144 different API functions [Viega 2000][4]. This was followed the next year by the release of two more tools, FlawFinder[5] and RATS[6]. FlawFinder, from David Wheeler, is a C/C++ scanning tool with a somewhat larger set of rules than ITS4. RATS, authored by John Viega, one of the original authors of ITS4, not only offered a broader ruleset covering 310 C/C++ API functions but also offered new rulesets for the Perl, PHP, Python, and OpenSSL domains. In parallel with this public development, Cigital, the company that originally created ITS4, released SourceScope, a follow-on to ITS4 with a new standard of coverage—653 C/C++ API functions.

Today there are around a half dozen first-tier options available in the static code analysis tools space. These tools include, but are not limited to, the following:

- Coverity Prevent[7]
- Fortify Source Code Analysis[8]
- Grammatech CodeSonar[9]
- Klocwork K7[10]
- Ounce Labs Prexis/Engine[11]
- Secure Software CodeAssure Workbench[12]

Each of these tools offers a comprehensive and growing ruleset varying in both size and area of focus. As you investigate and evaluate which tool is most appropriate for your needs, the coverage of the accompanying ruleset should be one of your primary factors of comparison.

This catalog contains full coverage of the C/C++ rulesets from ITS4, RATS, and SourceScope and is intended to represent the basic, foundational set of security rules for C/C++ development. Though there are currently tools available with rulesets much more comprehensive than this catalog, we consider this the **minimum standard** for any modern tool scanning C/C++ code for security vulnerabilities.

[3] http://www.cigital.com/its4/

[4] #refs

[5] http://www.dwheeler.com/flawfinder/

[6] https://securesoftware.custhelp.com/cgi-bin/securesoftware.cfg/php/enduser/doc_serve.php?2=Security

[7] http://www.coverity.com/products/products_security.html

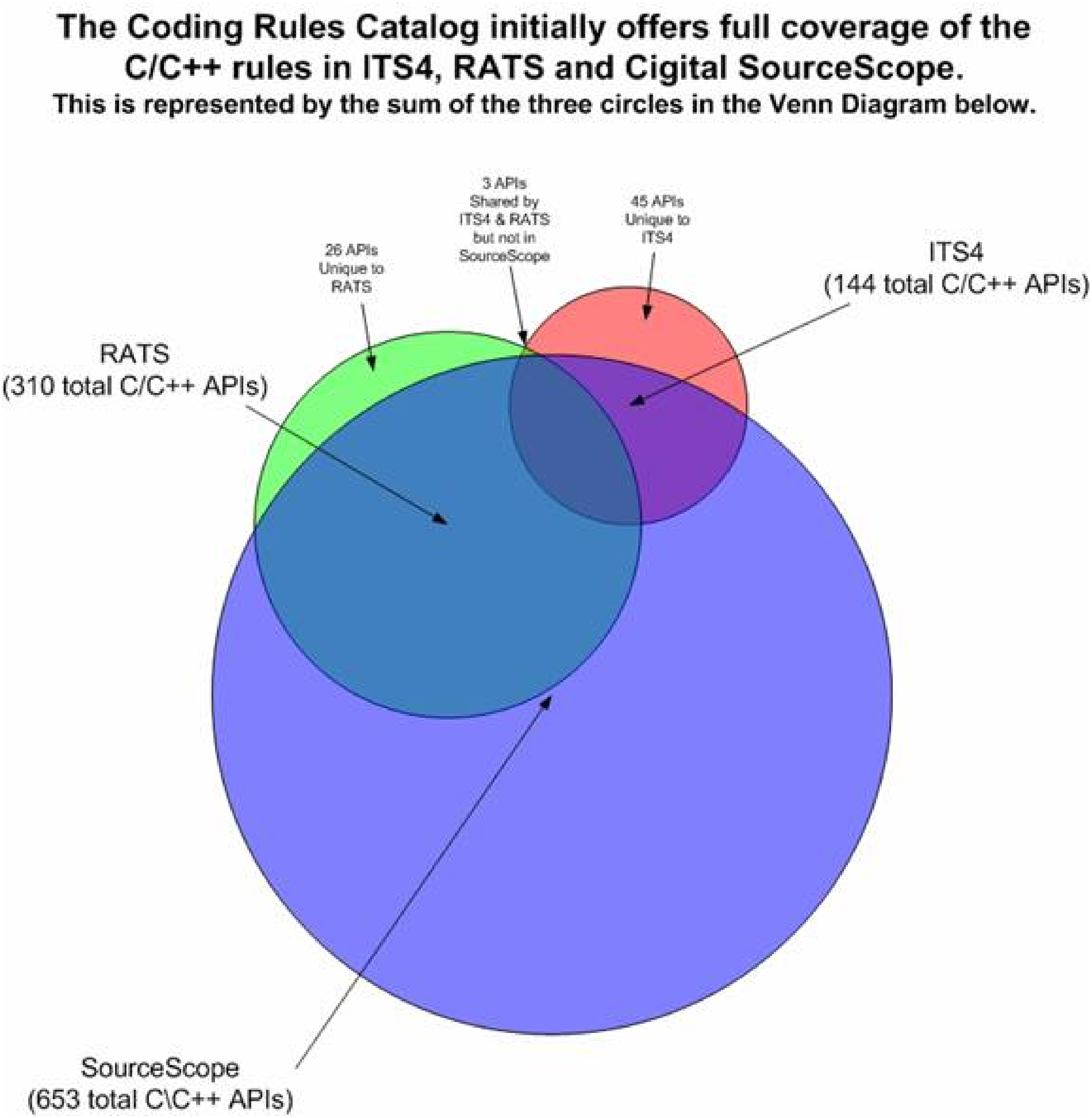
[8] http://www.fortifysoftware.com/products/sca/

[9] http://www.grammatech.com/products/codesonar/overview.html

[10] http://www.klocwork.com/products/klocworkk7.asp

[11] http://www.ouncelabs.com/prexis_engine.html

[12] http://www.securesoftware.com/products/source.html



The Coding Rules Catalog initially offers full coverage of the C/C++ rules in ITS4, RATS and Cigital SourceScope.
This is represented by the sum of the three circles in the Venn Diagram below.
3 APIs Shared by ITS4 & RATS but not in SourceScope
45 APIs Unique to ITS4
26 APIs Unique to RATS
ITS4 (144 total C/C++ APIs)
RATS (310 total C/C++ APIs)
SourceScope (653 total C\C++ APIs)

Evolution of Schema & Taxonomy

Just as the vulnerability coverage of rules has evolved over time, so have the organizational taxonomies used to manage them and the complexity of the schemas used to store them. First, let us clearly distinguish that in this discussion we are talking about coding rules and not just the vulnerabilities that they describe and expose. For many decades there have been scores of efforts to classify and categorize vulnerabilities. We leave that broader discussion for elsewhere. 1[13] Here we discuss specifically the context of coding rules and how their organization and descriptions have changed over time.

When coding rules were first explicitly described, they were simple lists of vulnerable APIs. Soon after collections of these APIs began to take form, people began to categorize them according to their simple

[13] #notes

vulnerability types, such as buffer overflow, time of check, time of use (TOCTOU), etcetera. These initial categorizations helped to identify achievable opportunities of focus for the first automated tools (e.g., Bishop and Dilger's race condition scanner).

Once more advanced tools entered the arena, we began to see information added to the rules beyond the simple API and category. For reporting purposes, including information such as a severity and a short description became necessary. These descriptions often included a recommended solution but were limited to only one or two lines of text all told. At this stage in the evolution of automated tools (during the era of ITS4, RATS, and FlawFinder), the value and novelty of the tools sprang from their ability to quickly point out potential issues because they had any *sort of list at all*. The user experience and depth of knowledge provided about various issues were both of lesser priority than the very existence of a tool.

This changed with the emergence of more capable and comprehensive tools like SourceScope. SourceScope pushed things forward for taxonomies and schemata in two ways. First, SourceScope was one of the first tools to move from considering every individual API a rule to aggregating sets of APIs that shared a common vulnerability and common solution in a smaller number of rules. A rule thus began to describe in a more useful way a potential vulnerability and its solution rather than simply unnecessary repetition across the numerous ways a vulnerability could occur. The underlying tool would still scan for all of the APIs, but would report a common set of information for all APIs covered by the same rule. Second, simplification and aggregation made it easier for SourceScope to use a more comprehensive schema and thus give the developer a much greater understanding of the potential vulnerability and how to mitigate or avoid it. This schema included more detailed descriptions and solutions, a description of the motivation of a likely attacker, estimated accuracy and severity, vulnerability category, references, and so on. A SourceScope-like comprehensive schema is now being pursued by the current players in the automated tool arena.

There are two main challenges remaining in the area of coding rules taxonomies and schemata. The first is that most rule schemata in use today are still too simple to convey the full context of a vulnerability to a degree that a developer can understand the issue that the tool is flagging and make wise choices in how to fix it. Simply pointing to a potential issue and saying “hey, go look over there” is inadequate for developers without a firm grounding in security. Unfortunately, those schemata that are becoming more comprehensive are typically proprietary and at odds with each other. We are working toward a common standard schema to describe software security coding rules. The second challenge is that rule taxonomies have started to grow out of control. They have moved from being bottom-up, well-defined structures to overly complex, theoretical exercises in what could possibly happen. This leads to unnecessary contention between models that disagree on the nature of the theoretical, as well as sparsely populated and less useful taxonomies. For this catalog, we decided to follow a strictly bottom-up philosophy in defining and populating our taxonomies. 2[14]

The table below presents the schema used for this catalog. It was derived from studying existing rule schemas and by discussing with experts in the field what sorts of information would make the rules more useful and meaningful.

Field Name	Field Description	Field Format	Selection Choices	Multiplicity
Title	Short rule descriptor	text		1
Attack Category	What typical types of attacks does this rule help expose and/or mitigate?	selection	Denial of Service Encryption Assault Environment Manipulation File Manipulation	*

[14] #notes

			Identity Spoofing Impersonation Malicious Input Memory Scanning Path Spoofing or Confusion Problem Privilege Exploitation Resource Injection	
Vulnerability Category	What types of vulnerabilities are exposed by this rule?	selection	Access Control Buffer Management Buffer Overflow Compiler Optimization Cryptography Format String Indeterminate File/Path Information Leakage Input Source (not really attack) Integer Overflow Multibyte Character No Null Termination Privilege Escalation Problem Process Management Race Condition Random Number Problems Temporary File Creation Problem Threading and Synchronization Problem TOCTOU (time of check, time of use) Unchecked Return Value Unconditional Unhandled Exception URL/Command Parsing	*
Sofware Context	In what area of software implementation does the rule have	set		*

	likely impact?			
• Context	Software implementation context of impact for this rule	selection	Authorization Critical Sections Cryptography Debug API ISAPI File Creation File Management Filename Management File Path Management File I/O Inheritance Internet Logging Memory Management National Language Support Networking Process Management Security Shell Functions String Formatting String Parsing String Conversion MACROS String Management Temporary File Management Threads and Processes	0..1
• Other Context	New software development contexts that are not in the Context list	text		0..1
Location	Header file, class, or module where this rule's APIs live	text		*
Description	Full explanation of the rule, things to search for, and potentially context of what can reduce the level of false positive hits on this rule	text		1

APIs	Which APIs is this rule applicable to?	set		*
• Function Name	API name	text		1
• Comments	Comments describing any special conditions of how this rule applies to this API	text		1
Method of Attack	Context/motivation of how this rule is important to an attacker. How would the attacker leverage this weakness to exploit the software?	text		1
Exception Criteria	Under what conditions is it okay to ignore the triggering of this rule?	text		1
Solution	What needs to be done to fix the code to avoid this rule and therefore improve the security of the code? What should be changed?	set		1..*
• Solution Applicability	The Solution Applicability is a natural language explanation of when it is appropriate to consider this solution.	text		1
• Solution Description	Description of the proposed actions or steps for this solution	text		1
• Solution Efficacy	The Solution Efficacy is a natural language explanation of the efficacy of this particular solution	text		1

Signature Details	What is the specific code signature to look for that will indicate that this rule is relevant for the code being analyzed?	text		1
Code Examples Negative	Specific code examples that exhibit this rule in failure mode. These examples are meant to simply illustrate the issue and are not intended to be compilable or usable in any sort of cut-and-paste fashion into any code. Every effort has been made to provide good quality examples, but their simple nature relies on human review for quality assurance. If you find any issues with the examples please let us know.	text		*
Code Examples Positive	Specific code examples that exhibit this rule in solution mode These examples are meant to simply illustrate the issue and are not intended to be compilable or usable in any sort of cut-and-paste fashion into any code. Every effort has been made to provide good quality examples, but their simple nature relies on	text		*

	human review for quality assurance. If you find any issues with the examples please let us know.			
Source References	Any supporting bibliography entries (sources) for this rule	text		*
Recommended Resources	Recommended resources for better understanding the context, nature, and implications of this rule	set		*
• Resource Name	Name of the resource being recommended	text		1
• Resource Link	URL link to the resource (if applicable)	text		1
Operating System	For which OS is this rule relevant?	set		1
• OS	Standard OS from list	selection	Any UNIX (All) UNIX Windows (All) Windows Windows 98 Windows Me Windows 2000 Windows XP Home Windows XP Pro Windows 2003 Windows Ce Win32 Palm OS Solaris HP-UX Linux AIX IRIX FreeBSD OpenBSD NetBSD MacOS X MacOS 9 Other	*

• Other OS	OS not in list	text		*
Language	For which programming language(s) is this rule relevant? (e.g., Java, C, C++, C#, VB)	text		*

The catalog published here takes the approach pioneered by SourceScope in aggregating APIs into common rules. This aggregation is typically defined by a commonality of both the underlying vulnerability as well as a somewhat specific solution. Because of the number of diverse APIs covered, the complexity of the schema, and the subjectivity of the aggregation criteria, some readers may feel that there is inconsistency in the aggregation applied in the catalog. We welcome any suggestions for improving this and any other dimension of the catalog.

As you can see from the schema, we have defined three primary dimensions of categorization for the rules: Attack Category, Vulnerability Category, and Software Context. Their current state is a direct reflection of the rules actually in the catalog. Of these, the most recognizable will be Vulnerability Category. There are currently many versions of this kind of taxonomy being put forward in the industry, including the one in the book *The 19 Deadly Sins of Software Security [Howard 05]* and the one in *Software Security: Building Security In* by Gary McGraw (to be published by Addison-Wesley in February 2006). It is essential that the community works together toward a standard vulnerability category taxonomy to enable greater consistency of discussion as well as portability of categorized knowledge such as coding rules. We welcome the emergence of a standard.

One result of the categorization of the rules is the initial creation of some MetaRules defining a higher level, more abstract rule for some typical groupings of rules in the catalog. These MetaRules provide a developer with a good overview of various types of vulnerabilities, their solutions, and which particular individual APIs are susceptible to them. This effort has only begun and will continue with the evolution of the catalog as new MetaRules are created and others are split or merged as needed. Eventually, each MetaRule will also be bidirectionally linked with its relevant individual rules.

At the top and bottom of this Coding Rules Catalog introduction, you are provided with the ability to search or browse the catalog in several ways, including by the various taxonometric categories.

Objectives of the Coding Rules Catalog

This catalog of coding rules is presented here as part of the overall Building Security In knowledge content with several objectives in mind.

- **To act as measuring stick for evaluating software security code analysis tools.** There are numerous choices of both open-source and commercial tools available today for software security code analysis. These tools include, but are not limited to:
    - Coverity Prevent[15]
    - Fortify Source Code Analysis[16]
    - Grammatech CodeSonar[17]
    - Klocwork K7[18]
    - Ounce Labs Prexis/Engine[19]
    - Secure Software CodeAssure Workbench[20]

  These tools are the best way to apply the coding rules in practice. Recognizing that all situations are not equal and that there is not any one tool that is right for everyone, this catalog is presented to give

[15] http://www.coverity.com/products/products_security.html

[16] http://www.fortifysoftware.com/products/sca/

[17] http://www.grammatech.com/products/codesonar/overview.html

[18] http://www.klocwork.com/products/klocworkk7.asp

[19] http://www.ouncelabs.com/prexis_engine.html

[20] http://www.securesoftware.com/products/source.html

you an objective reference of measure to compare available tools based on the most important dimension of their capability—the rulesets that they check for. Click here[21] to learn more about Code Analysis Tools.

We understand that there are commercial tools currently available that offer rulesets more comprehensive than the one making up this catalog. This catalog is intended to represent the *minimum level of coverage* that should be available in a modern C/C++ language security code analysis tool. If your tool does not support this basic ruleset, you should consider an alternative. Currently, the catalog will be useful only for those comparing tools that review C/C++ code. In the future, the catalog will be expanded to cover other languages. For more information on selecting software security code analysis tools, please see the Code Analysis Tools[22] section of the Building Security In website.

- **To act as an instructive reference for developers so they can understand vulnerabilities that current tools flag.**
  Even among the most modern tools with comprehensive rulesets and advanced analysis technologies, there is often little information offered to the user about any particular issue being flagged. There is a need for more information to help put the flagged issue in context and to help the user decide on the most effective mitigating action to take. This would be a mitigation that not only clears the flagging by the tool but also resolves the underlying security vulnerability in question. This catalog is intended to act as such a reference for the rules that it contains. It also provides an excellent example to tool developers of the kinds of information that should be integrated into the tools in the future.
- **To act as an instructive reference for developers so they can learn about potential vulnerabilities and how to avoid them up front before a tool flags them.**
  In the spirit of *Building Security In* , the catalog is also intended as a resource of knowledge for developers to explore proactively as a learning exercise to better understand the vulnerabilities that exist in their source language so that they can avoid them in the first place. As we *move left* in the life cycle in mitigating the risk of potential vulnerabilities in our code, we reduce our overall effort and greatly reduce the impact this risk has on our software.
- **To act as a catalyst to spur conversation and collaboration among the software security community around coding rules.**
  This catalog is a continuously evolving and growing body of knowledge that not only absorbs the knowledge and input of the practitioners in the community but also challenges them to expand their own understanding and expectations around the topic of security coding rules. Healthy dynamic growth will not be sustained with only the participation of the group of experts who compiled this initial version. It will require the deep involvement of the users of this website. Please offer your experience and insight. As we go forward here, keep your eyes open for more collaborative capabilities to be added to the website to enable your more dynamic involvement.

Potential Next Steps for the Coding Rules Catalog

Recognizing that this catalog is intended to be a continuously evolving and growing body of knowledge, here are a few next steps under consideration (and in some cases in development) for the catalog in the future:

- referential linking between related rules and other BSI website content
- ongoing enhancement of current rules in the catalog, such as refinements to the current descriptions and solutions, new solutions added, new code examples, etc.
- new rules in the C/C++ language base
- # new rules for other languages such as Java, .NET, PHP, shell, AJAX, etc.
- potential extensions to the schema, including updating current rules. These extensions could take the form of compilable code examples to use as test suites for tools, normalized signature definition

[21] https://buildsecurityin.preview.us-cert.gov/portal/article/tools/Code_Analysis

[22] https://buildsecurityin.preview.us-cert.gov/portal/article/tools/Code_Analysis

using a standard expression language, extensions required for new types of rules like data flow, etc.

Acknowledgements and Disclaimers

Acknowledgements

Coding Rule Pioneers : Matt Bishop, Michael Dilger, John Viega, Gary McGraw, and David Wheeler. The Cigital rule building team included Frank Charron, Mike Debnam, John Steven, J. Richard Mills, Viren Shah, and Chris Ren.

This list is by no means all inclusive, but it was the work of these gentlemen that inspired this catalog.

Internal team of authors, reviewers & contributors : Sean Barnum, Paco Hope, Will Kruse, Michael Gegick, Robert Wentworth, Joseph Wisniewski, Amit Sethi, Fabio Arciniegas, Gary McGraw

External expert reviewers : Fred Schneider (Cornell University), Brian Chess (Fortify), Steve Lipner (Microsoft), Michael Howard (Microsoft), Shawn Hernan (Microsoft), John Viega (Secure Software).

Disclaimer

This catalog of rules is not a completely original creation, but rather an original presentation, aggregation and collection of knowledge from hundreds of people and sources. Its value lies not in its originality of content but rather in the novelty of presenting such a diverse set of knowledge together in one place and in a unified, searchable context. The authors of this catalog have done their best to cite and quote other work that was used in the aggregation effort; however, with such a large set of content and diversity of sources, it is possible that we inadvertently missed something. If you notice any oversights please let us know and we will fix them.

Footnotes

[1] For a good start, see Ivan Krusl's thesis A Taxonomy of Security Faults in the Unix Operating System (1995) from CERIAS (then called COAST) http://homes.cerias.purdue.edu/~spaf/students.html[23].

[2] Because of this, an experienced reader may quickly notice some obvious categories appear to be missing compared to other modern taxonomies. This is purely a result of the rules in this catalog and not a statement of what the complete version could or should be.

References

[Bishop 96] Bishop, Matt & Dilger, Mike. “Checking for Race Conditions in File Accesses.” Computing Systems 9, 2 (1996): 131–152.

[Viega 2000] Viega, John; Bloch, J. T.; Kohno, Tadayoshi; & McGraw, Gary. "ITS4: A Static Vulnerability Scanner for C and C++ Code." *Proceedings of Annual Computer Security Applications Conference*. New Orleans, LA, December 11-15, 2000. http://www.acsac.org/.

Cigital, Inc. Copyright

[23] http://homes.cerias.purdue.edu/%7Espaf/students.html

####

###	########
Copyright Holder	Cigital, Inc.

####

###	########
is-content-area-overview	true
Content Areas	Knowledge/Coding Rules
Workflow State	Publishable

[1] mailto:copyright@cigital.com